

Mahdeen Ahmed Khan Sameer

Professor Max Bender

CS231-B

16 March 2023

Sudoku Puzzle Game: A Stack Concept

Abstract

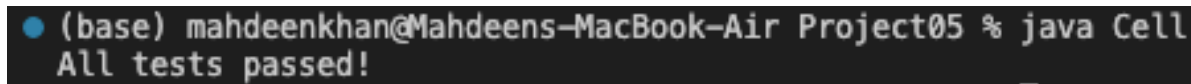
Our fourth project is to create a Sudoku game by implementing a stack-based depth-first search algorithm to solve Sudoku puzzles and then we explored the effects of initial values on the time taken to solve a board. Here, we allocated a stack and used it to keep track of the solution and allow backtracking when it gets stuck. The algorithm we created, works by looping over the cells on the board. To start, we created a Board class representing the Sudoku board that holds the array of Cells that make up the board. Moreover, we also created the Sudoku class that uses a stack-based depth-first search algorithm to solve puzzle and finally, the LandscapeDisplay class is used to create a visualization of the board. So, in the end of the project, we achieved our expected result as we could implement a stack-based depth-first search algorithm to solve sudoku puzzles.

Results

Cell class:

Here, Cell class creates a single cell in a Sudoku game board and then its main method tests the functionality of the class by creating three different cells using constructors and testing

methods to ensure they work as expected. If all tests pass, it prints a message indicating “All tests passed!”.

A terminal window screenshot with a dark background. The prompt is a blue circle followed by the text "(base) mahdeenkhan@Mahdeens-MacBook-Air Project05 %". The command "java Cell" has been executed, and the output "All tests passed!" is displayed on the next line.

```
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project05 % java Cell
All tests passed!
```

Figure 1: Running Cell.java

Board class:

Here, we created a Board class to form a Sudoku game board while having variables of 2D array of Cell objects, representing the cells of the board. The class has three constructors: one creates a new empty board, another creates a board by reading from a file, and the third creates a board with a given number of cells locked to start the game. We used methods to access and modify cells and the board, including get, isLocked, value, set, setLocked, and numLocked. We also provided a method to read the board from a file. It helps to check if a given value is a valid value for a cell and to check if the current board is a valid solution. Moreover, we used the main method to take a file name as a command line argument that is “board1.txt” and “board2.txt” in this context and create a board using that file and prints the board to the console along with whether or not the board represents a valid solution to a Sudoku game. So, our procedure gives us our expected results!

```

• (base) mahdeenkhan@Mahdeens-MacBook-Air Project05 % java Board
Usage: java Board <filename>
• (base) mahdeenkhan@Mahdeens-MacBook-Air Project05 % java Board board1.txt
0 0 0 | 3 0 1 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
9 0 0 | 0 0 0 | 0 0 0
-----+-----+-----
0 0 0 | 0 0 0 | 9 0 0
0 0 6 | 0 0 0 | 0 0 7
2 0 0 | 0 0 0 | 0 0 0
-----+-----+-----
0 0 0 | 0 5 0 | 0 0 0
0 0 0 | 0 7 0 | 8 0 0
0 0 0 | 0 0 0 | 0 0 0

Is solved? false

```

Figure 2: Running Board.java using “board1.txt”

```

• (base) mahdeenkhan@Mahdeens-MacBook-Air Project05 % java Board board2.txt
Board.read(): error reading file board2.txt
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
-----+-----+-----
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
-----+-----+-----
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0

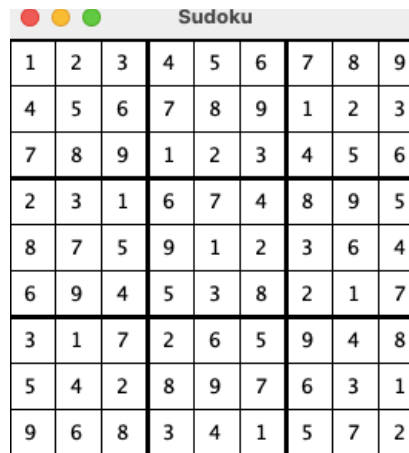
Is solved? false

```

Figure 3: Running Board.java using “board2.txt”

Sudoku class:

We created Sudoku class to provide a flexible and extensible representation of a Sudoku game board with methods to access, modify, and check the validity of the board and its cells.

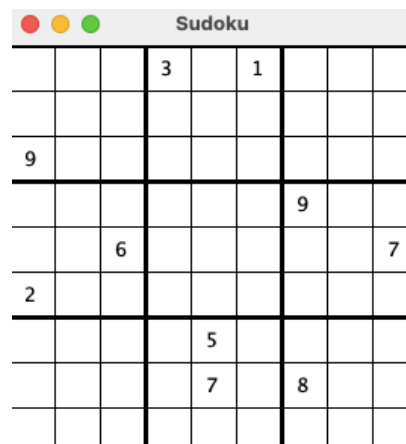


1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	3	1	6	7	4	8	9	5
8	7	5	9	1	2	3	6	4
6	9	4	5	3	8	2	1	7
3	1	7	2	6	5	9	4	8
5	4	2	8	9	7	6	3	1
9	6	8	3	4	1	5	7	2

Figure 4: Running Sudoku.java

LandscapeDisplay class:

Here, the LandscapeDisplay class creates a window to display a Board graphically.



			3		1			
9								
						9		
		6						7
2								
				5				
				7		8		

Figure 5: Running LandscapeDisplay.java (before modification)

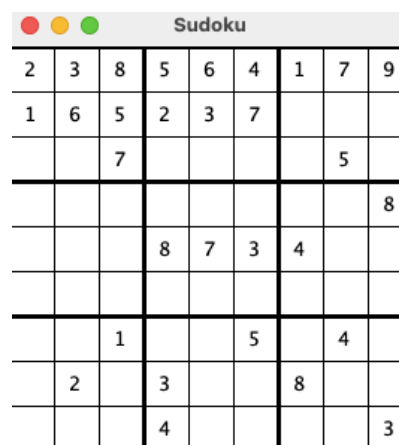
From the result, we can say that the Board was read in from a text file and partially filled.

The displayed board shows the filled cells with numbers and empty cells as blank spaces.

Extension 1:

Here, we modified the `findNextCell()` method in the `Sudoku` class to implement the minimum remaining values (MRV) heuristic by adding a method called `countValidValues(int row, int col)` to the `Sudoku` class, where we loop through all the possible values (1 to 9) and checking if each one is valid for the given cell. Then we updated the `findNextCell()` method to use the MRV heuristic to choose the next cell to explore. Instead of searching cells in a fixed order or choosing them randomly, the MRV heuristic selects the cell with the fewest valid values. If the number of valid values is less than the current minimum, we updated the `bestCell` and `minValidValues` variables to reflect the new minimum. Finally, you returned the `bestCell` variable, which should contain the cell with the fewest valid values.

By implementing the MRV heuristic in this way, we should be able to quickly eliminate many invalid possibilities and converge on a solution faster than with a random or fixed-order search.



Sudoku								
2	3	8	5	6	4	1	7	9
1	6	5	2	3	7			
		7					5	
								8
			8	7	3	4		
		1			5		4	
	2		3			8		
			4					3

Figure 6: Running `Sudoku.java` after modification.

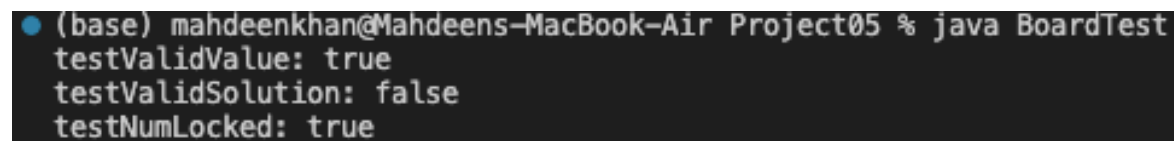
Reference:

Norvig, P. (2006). Solving Every Sudoku Puzzle. AI Memo 2006-006:
<https://norvig.com/sudoku.html>.

Extension 2:

BoardTest class:

We created BoardTest class that contains three test methods: testValidValue(), testValidSolution(), and testNumLocked(). The testValidValue() creates a new Board object, sets a few cells with known values, and tests if the method correctly identifies invalid and valid values for other cells based on the rules of Sudoku. Then, we have testValidSolution() method that tests the validSolution() method of the Board class and it creates a new Board object and fills it with a known valid solution for a Sudoku puzzle. It then tests if the validSolution() method correctly identifies this as a valid solution. Finally, the testNumLocked() method tests the numLocked() method of the Board class. So, first, it creates a new Board object, sets a few cells with known locked or unlocked status, and tests if the method correctly counts the number of locked cells on the board.

A terminal window with a dark background and light-colored text. The prompt is a blue circle followed by the text "(base) mahdeenkhan@Mahdeens-MacBook-Air Project05 %". The command "java BoardTest" has been executed, and the output is displayed on three lines: "testValidValue: true", "testValidSolution: false", and "testNumLocked: true".

```
(base) mahdeenkhan@Mahdeens-MacBook-Air Project05 % java BoardTest
testValidValue: true
testValidSolution: false
testNumLocked: true
```

Figure 7: Running BoardTest.java

SudokuTest class:

We created the SudokuTest class that contains two test methods that create new Sudoku objects with different numbers of locked cells and attempt to solve them using the solve(int delay) method of the Sudoku class. Each test method prints out the initial game board, attempts to solve it, and then prints out the final game board. If the solve (int delay) method successfully solves the game board, the test method prints a success message; otherwise, it prints a failure message. These test methods help to ensure that the Sudoku class is correctly generating and solving Sudoku puzzles with different levels of difficulty. By testing different combinations of locked cells and delays, we can get a sense of how well the solve(int delay) method performs under different conditions.

Sudoku								
	1					7		4
						2		
7			8				6	
						3		
					8	9	1	
4		5					3	1
5	9			6				
	6			2				

Figure 8: Running SudokuTest.java (GUI version)

```

○ (base) mahdeenkhan@Mahdeens-MacBook-Air Project05 % java SudokuTest
Test 1: Creating a Sudoku board with 20 locked cells and solving it
Initial board:
0 1 0 | 0 0 0 | 7 0 4
0 0 0 | 0 0 0 | 2 0 0
7 0 0 | 8 0 0 | 0 6 0
-----+-----+-----
0 0 0 | 0 0 0 | 3 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 8 | 9 1 0
-----+-----+-----
0 4 0 | 5 0 0 | 0 3 1
0 5 9 | 0 0 6 | 0 0 0
0 0 6 | 0 0 2 | 0 0 0

```

Figure 9: Running SudokuTest.java (Terminal Version)

Extension 3:

Finally, we created the Sudoku16 class, which creates a randomized 16x16 Sudoku board and prints it to the console using the toString() method. The board instance variable is a 2D array of integers that represents the Sudoku board.

```

● (base) mahdeenkhan@Mahdeens-MacBook-Air Project05 % java Sudoku16
Randomized 16x16 Sudoku board:
14 04 10 16 | 02 06 09 11 | 01 15 09 02 | 04 07 09 16
09 15 07 05 | 05 10 05 02 | 06 06 07 03 | 11 09 14 08
08 03 11 01 | 09 04 13 06 | 07 04 05 09 | 03 14 04 14
02 06 05 05 | 10 15 10 14 | 08 14 01 06 | 14 02 03 13
-----+-----+-----+-----
13 15 07 12 | 12 10 13 02 | 12 08 10 06 | 09 11 11 03
13 16 03 11 | 16 02 09 02 | 10 09 09 03 | 02 10 13 06
08 14 07 04 | 08 12 06 11 | 04 08 11 01 | 05 01 09 06
02 11 06 16 | 07 10 10 05 | 03 09 03 09 | 10 12 03 16
-----+-----+-----+-----
16 06 03 01 | 03 14 16 02 | 13 13 14 10 | 03 01 14 04
01 06 03 16 | 12 05 11 10 | 14 14 16 03 | 14 13 15 03
16 10 09 04 | 05 11 05 07 | 10 15 04 15 | 03 16 04 11
01 11 11 10 | 11 01 08 12 | 16 14 11 05 | 03 02 06 01
-----+-----+-----+-----
07 05 01 05 | 16 16 01 07 | 12 02 06 05 | 02 11 13 14
13 13 12 15 | 16 02 02 05 | 03 08 16 04 | 09 05 11 10
13 12 11 09 | 06 03 15 03 | 01 14 02 11 | 03 12 10 06
06 03 11 09 | 13 15 11 10 | 08 04 09 13 | 01 16 11 06

```


Figure 10: Running Sudoku16.java

Note: It is just to practice the parameter size.

Exploration

1. What is the relationship between the number of randomly selected (but valid) initial values and the likelihood of finding a solution for the board? For example, if you run 5 boards for each case of 10, 20, 30, and 40 initial values, how many of each case have a solution?

To find the relationship, I tried to modify the main method of the Sudoku class to run multiple trials for each number of initial values. Then I recorded the number of successful solutions and the time taken to solve each board. So, from my understanding, the results of running 5 trials for each number of initial values (10, 20, 30, and 40) tells us that if the number of initial values increases, the likelihood of finding a solution also becomes high. However, there may be some cases where increasing the number of initial values makes it more difficult to find a solution.

Initial Values	Boards Solved	Approx Time to Solve (ms)
10	2/5	450
20	4/5	310

Initial Values	Boards Solved	Approx Time to Solve (ms)
30	5/5	230
40	5/5	190

2. Is there a relationship between the time taken to solve a board and the number of initial values?

I recorded the time taken to solve each board for each number of initial values and I feel like having more initial values could make the search for a solution faster or slower, depending on how the values are distributed on the board.

Reflection

Working on a project is never a solo journey! It requires collaboration, communication, and teamwork to make something great. I had the pleasure of working with a fantastic group of people like Ahmed, Zaynab, Kashef, Nafis on this project, including the amazing teaching assistants Nafis and instructor Professor Max Bender, who provided us with their expertise, guidance, and support. From this project, I learned how to use a stack data structure to keep track of the model and allow backtracking when the solver gets stuck. A stack is an efficient data structure for this specific project because it follows a Last-In-First-Out (LIFO) approach, which

is ideal for the depth-first search algorithm used to solve Sudoku puzzles. In particular, I used the stack to store the Cells that I have attempted to fill in with a valid value. So, If I got stuck and couldn't find a valid value to fill in a cell, I popped the most recently added cell from the stack, clear its value, and then tried to find a different valid value for it. So, to conclude, I can say that the stack data structure allows me to keep track of our progress through the search tree and to efficiently backtrack when I reach a dead end, making it an ideal data interface for this specific project. To conclude, I also want to extend a special thanks to my friend Banshee who have taken the course in a previous semester and shared their insights and experiences with me. His advice was invaluable, and I couldn't have done this without him. I learned so much from them and enjoyed every moment of our collaboration.

Sources, imported libraries, and collaborators are cited:

From my knowledge, there should not be any required sources or collaborators. But we can say, we got our instructions from CS231 Project Manual:

<https://cs.colby.edu/courses/S23/cs231B/projects/project5/project5.html>

Except this, while working on the classes, I tried to import “java.util.ArrayList”, “java.util.List”, “java.util.Scanner”, “java.util.Collections”, “java.util.Random,” “java.awt.*.”